# The Equalizer Parallel Rendering Framework

Stefan Eilemann* Renato Pajarola†

Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

*Technical Report IFI-2007.06, Department of Informatics, Unviersity of Zürich*

**ABSTRACT**

Continuing improvements in CPU and GPU performances as well as increasing multi-core processor and cluster-based parallelism demand for scalable parallel rendering solutions that can exploit multipipe hardware accelerated graphics. In fact, to achieve interactive visualization, scalable rendering systems are essential to cope with the rapid growth of data sets. However, parallel rendering solutions are non-trivial to develop and often only application specific implementations have been proposed. The task of developing a scalable parallel rendering framework is even more difficult if it should be generic to support various types of data and visualization applications, and at the same time work efficiently on a cluster with distributed graphics cards.

In this paper we introduce Equalizer, a toolkit for scalable parallel rendering based on OpenGL which provides an application programming interface (API) to develop scalable graphics applications for a wide range of systems ranging from large distributed visualization clusters and multi-processor multipipe graphics systems to single-processor single-pipe desktop machines. We describe the architecture of Equalizer, discuss its advantadges over previous approaches, present example configurations and usage scenarios as well as some scalability results.

**Keywords:** Scalable Rendering, Parallel Rendering, Immersive Environments, Scalable Graphics Hardware.

## 1 INTRODUCTION

The continuing improvements in hardware integration lead to ever faster CPUs and GPUs, as well as higher resolution sensor and display devices. Moreover, increased hardware parallelism is applied in form of multi-core CPU workstations, massive parallel super computers, or cluster systems. Hand in hand goes the rapid growth in complexity of data sets from numerical simulations and high-resolution 3D scanning systems, which causes interactive exploration and visualization of such large data sets to become a serious challenge. It is thus crucial for a visualization solution to take advantage of hardware accelerated scalable parallel rendering. While the framework presented in this paper works as well in a shared-memory system, we focus more on cluster-parallel rendering, because workstation graphics hardware is developing faster than high-end (super-) computer graphics, thereby outperforming such integrated solutions.

Previous parallel rendering approaches typically failed in one of the following ways: a) provided only special domain solution, b) offered transparent, but not scalable abstraction of the graphics layer, or c) required replacing most of the existing code infrastructure (e.g. such as proprietary scene graphs, molecular data structures, level-of-detail and geometry databases). To date, generic and scalable parallel rendering frameworks that can be adopted to a wide range of scientific visualization domains are not yet readily available. Furthermore, flexible and automatic configurability to arbitrary cluster and display-wall configurations has also not been addressed in the past, but is of immense practical importance to scientists using high-performance interactive visualization as a scientific tool. In this paper we present *Equalizer*, which is a novel flexible framework for parallel rendering that supports scalability, is *minimally invasive* with respect to adapting existing visualization applications, and applies to virtually any scientific visualization application domain.

Equalizer is open source, available under the LGPL license from [13], which allows it to be used both for open source and commercial applications. It is very portable, and has been tested on Linux, Microsoft Windows, and Mac OS X in 32 and 64 bit mode using little endian and big endian processors.

*e-mail:eilemann@gmail.com
†e-mail:pajarola@acm.org

## 2 RELATED WORK

The early fundamental concepts of parallel rendering have been laid down in [38] and [12]. A number of domain specific parallel rendering algorithms and special-purpose hardware solutions have been proposed in the past, however, only few generic parallel rendering frameworks have been developed.

### Domain specific solutions

Cluster-based parallel rendering has been commercialized for off-line rendering (i.e. distributed ray-tracing) for computer generated animated movies or special effects, since the ray-tracing technique is inherently amenable to parallelization for off-line processing. Other special-purpose solutions exist for parallel rendering in specific application domains such as volume rendering [34, 53, 23, 48, 19, 43] or geo-visualization [52, 2, 33, 29]. However, such specific solutions are typically not applicable as a generic parallel rendering paradigm and do not translate to arbitrary scientific visualization and distributed graphics problems.

### Special-purpose architectures

Traditionally, high-performance real-time rendering systems have relied on an integrated proprietary system architecture, such as the SGI graphics super computers. These special-purpose solutions have become a niche product as their graphics performance does not keep up with off-the-shelf workstation graphics hardware and scalability of clusters. However, cluster systems need more sophisticated parallel graphics rendering libraries, such as the one proposed in this paper.

Due to its conceptual simplicity, a number of special-purpose image compositing hardware solutions for sort-last parallel rendering have been developed. The proposed hardware architectures include Sepia [37, 32], Sepia 2 [35, 36], Lightning 2 [49], Metabuffer [8, 56], MPC Compositor [42] and PixelFlow [39, 18], of which only a few have reached the commercial product stage (i.e. Sepia 2

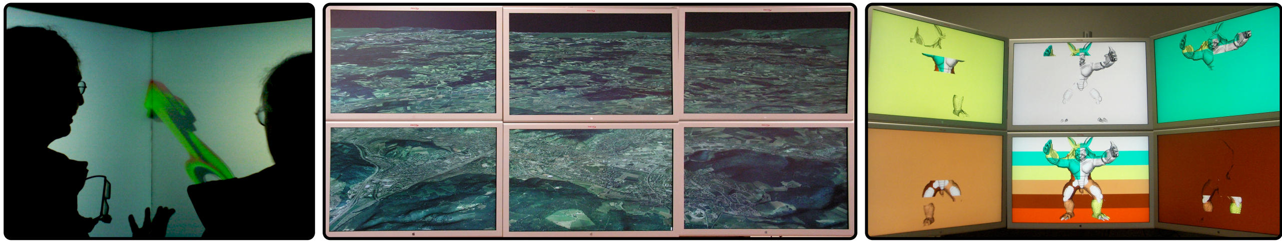


Figure 1: Various Equalizer use cases.

and MPC Compositor). However, the inherent inflexibility and setup overhead have limited their distribution and application support. Moreover, with the recent advances in the speed of CPU-GPU interfaces, such as PCI Express and other modern interconnects, combinations of software and GPU-based solutions offer more flexibility at comparable performance.

### Generic approaches

A number of algorithms and systems for parallel rendering have been developed in the past. On one hand, some general concepts applicable to cluster parallel rendering have been presented in [40, 41] (sort-first architecture), [47, 46] (load balancing), [45] (data replication), or [10, 9] (scalability). On the other hand, specific algorithms have been developed for cluster based rendering and compositing such as [3], [11] and [54, 50]. However, these approaches do not constitute APIs and libraries that can readily be integrated into existing visualization applications, although the issue of the design of a parallel graphics interface has been addressed in [28]. Only few generic APIs and (cluster-) parallel rendering systems exist which include VR Juggler [7] (and its derivatives), Chromium [27] (an evolution of [26, 24, 25]) and OpenGL Multipipe SDK [30, 5, 1].

VR Juggler [7, 31] is a graphics framework for virtual reality applications which shields the application developer from the underlying hardware architecture, devices and operating system. Its main aim is to make virtual reality configurations easy to set up and use without the need to know details about the devices and hardware configuration, but not specifically to provide scalable parallel rendering. Extensions of VR Juggler, such as for example ClusterJuggler [6] and NetJuggler [4], are typically based on the replication of application and data on each cluster node and basically take care of synchronization issues, but fail to provide a flexible and powerful configuration mechanism that efficiently supports scalable rendering.

The main limitation of Chromium [27] for scalable rendering is that it is focused on streaming OpenGL commands through a network of nodes, often initiated from a single source. The problem comes in that the OpenGL stream can be very large in size, not only containing OpenGL calls but also all the rendered data such as geometry and texture images. Furthermore, this stream of function calls and data must be packaged and broadcast in real-time over the network to multiple nodes for each rendered frame. This makes CPU performance and network bandwidth a major limiting factor. But for high-performance visualization of large-scale data it is immensely important to limit real-time data distribution over the network.

OpenGL Multipipe SDK (MPK) [5] implements an effective parallel rendering API for a shared memory multi-CPU/GPU system. It is similar to IRIS Performer [44] in that it handles multipipe rendering by a lean abstraction layer via a conceptual callback mechanism, and that it runs different application tasks in parallel. However, MPK is not designed nor meant for rendering nodes separated by a network. MPK focuses on providing a parallel rendering framework for a single application, parts of which are run in parallel on multiple rendering channels, such as the culling, rendering and final image compositing processes. Unlike Chromium, it is not fully transparent but minimally invasive with respect to changes to existing visualization applications. This concept enables scalable high-performance rendering while at the same time protecting the main customer investments into proprietary code infrastructure, and this approach is also taken in Equalizer.

## 3 Basic Concepts

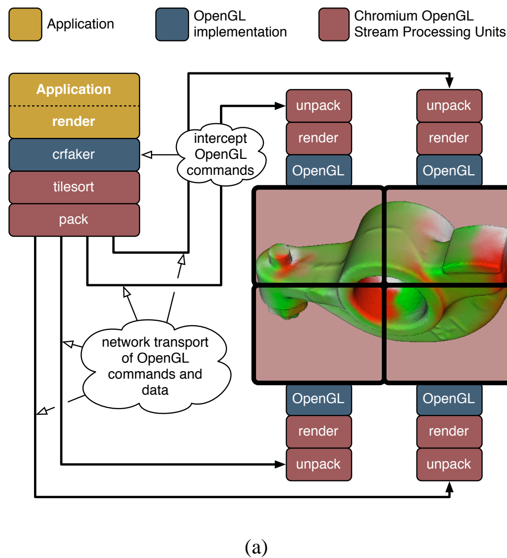
Besides the API, one of the major differences of Equalizer to Chromium is that it generally distributes and runs the application code in parallel. For example, one can setup a multi-screen display-wall with Chromium, streaming the OpenGL calls to a number of render nodes assigned to screen tiles of the display-wall, as illustrated in Figure 2(a). One instance of the application is running. In contrast, Equalizer runs parts of the application in parallel on multiple rendering channels as illustrated in Figure 2(b).

Equalizer takes care of distributed execution, synchronization and final image compositing, while the application programmer identifies and encapsulates critical parts of the application, such as culling and rendering. This approach is considered to be *minimally invasive* since the existing rendering code can basically be retained. The minimal change needed for Equalizer is that the application rendering code uses the frustum parameters, viewport and stereo buffer provided by Equalizer for rendering. The application should implement efficient view frustum culling for performance, in particular for sort-first decompositions. For sort-last rendering, the application should support rendering a subset of the application-specific database, given by a one-dimensional *range* interval. Hence network bandwidth is freed from unnecessary transmission of excessive graphics commands and data since only the basic rendering parameters are exchanged between nodes. Only for the unavoidable final image compositing step in scalable rendering Equalizer exchanges framebuffer data between the nodes.

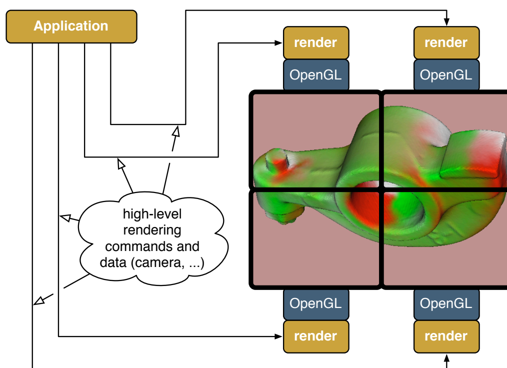
A major strength of Equalizer is its flexible and scalable configuration of the parallel rendering tasks, which takes the notion of a compound tree introduced in MPK [5] to a distributed cluster environment as discussed in the following section. Hence different parallel rendering task decomposition and image compositing configurations can easily be specified, see also Figure 8. For example, efficient sort-last image compositing has been demonstrated in [17].

The Equalizer framework does not impose any constraints on how the application handles and accesses the data to be visualized. As such, Equalizer does not provide a solution to the parallel data access and distribution problem which has to be addressed by the application itself, for example via mechanisms to limit data replication (e.g. [45]), or out-of-core access to large data sets and multiresolution representations (e.g. [11]). As demonstrated in [11], out-of-core data structures are well suited to provide efficient parallel access to the 3D data from all rendering nodes, and a wealth of out-of-core approaches have been provided for volume, polygonal or point data sets (e.g. [51], [22], [55], [21] or [20]). Equalizer does not interfere with or inhibit any solution to this problem, as it is an orthogonal issue.

Equalizer does address some fundamental problems to help application developers to distribute their data effectively in the context of parallel rendering. The Equalizer networking layer supports




(a)




(b)

Figure 2: (a) A typical Chromium setup, and (b) an Equalizer application both driving a 2x2 display wall.

message passing and the creation of distributed objects. By subclassing a distributed object class, static and versioned objects can be created. Objects are addressed on the cluster using a unique identifier, which allows the remote mapping of the object. Versioned objects are typically used for frame-specific data, where a new version for each new frame is created. This version information is passed correctly by Equalizer to the application rendering code. This mechanism allows simple distribution and multi-buffering of data.

## 4 System Architecture

### 4.1 Interface

Equalizer is a parallel rendering framework using a similar underlying concept as OpenGL Multipipe SDK (MPK). In the following we will focus on the architectural improvements of Equalizer compared to MPK. The configuration interface to specify the hardware setup and task decomposition, following the nomenclature of MPK [5], is further described in Section 4.5.

Equalizer provides a framework to facilitate the development of distributed and non-distributed parallel rendering applications. The programming interface is based on a set of C++ classes, modeled closely to the hierarchical resource description used by the server. The application subclasses these objects and overrides C++ task methods, similar to C callbacks. These task methods will be called in parallel by the framework, depending on the current configuration. A wrapper interface could be written to provide C bindings.

In contrast to MPK, an Equalizer application does not select the rendering configuration itself; it is configured by a system-wide configuration server. The application is written against a client library, which communicates with the server. The configuration is choosen by the server based on guidelines from the application or user. The server also launches and controls the rendering clients provided by the application.

While on a higher level Equalizer uses a client-server approach, it is built on a peer-to-peer network layer. This network layer provides a message-based communication interface, as needed between any two nodes in the cluster, e.g., to transmit image data for result recomposition during scalable rendering. Currently Equalizer provides an implementation for TCP/IP sockets and InfiniBand. The usage of MPI as a low-level communication library was not feasible in the context of Equalizer. Dynamic process management is only available in MPI 2, which still is not wide-spread enough. Furthermore, the communication patterns for which MPI was designed are significantly different from Equalizer's use case. However, this does not prohibit coupling MPI-based programs with Equalizer.

### 4.2 Application

The application in Equalizer solely drives the rendering, that is, it carries out the main rendering loop only, but does not actually execute any rendering. Although depending on the configuration, the application process may also host one or more render client threads, as described below. When a configuration has no additional nodes besides the application node, all application code is executed in the same process, and no network data distribution has to be implemented. In this special case, an Equalizer application is very similar to a MPK application.

During initialization of the server, the application provides a rendering client. The rendering client is often, especially for simple applications, the same executable as the application. However, the rendering client may be a thin renderer which only contains the application-specific rendering code. The server deploys this rendering client on all nodes specified in the configuration. The main rendering loop is quite simple: The application requests a new frame to be rendered, synchronizes on the completion of a frame and processes events received from the render clients. Figure 3 shows a simplified execution model of an Equalizer application.

### 4.3 Rendering Client

Each Equalizer application provides a rendering client, which can be the same executable as the application itself. In contrast to the application, the rendering client does not need a main loop and is completely controlled by the Equalizer framework. Some configurations use the application process as a node, in which case the rendering happens in a different thread within the application process. A render client consists of the following threads: The node main thread, one network receive thread, and one pipe thread for each pipe to execute rendering tasks.

The client library implements the main loop, which receives network events and processes them. Most importantly, the network data contains the rendering task parameters computed by the server. Based on this data, the client library sets up the rendering context and calls the application-provided task methods. Setting up the rendering context consists of using the correct rendering thread, making the drawable and context current, as well as providing the task methods with the 2D viewport, frustum, view matrix and the data-range for sort-last rendering. The task methods clear the frame buffer as necessary, execute the OpenGL rendering commands as well as readback, and assemble partial frame results for scalable

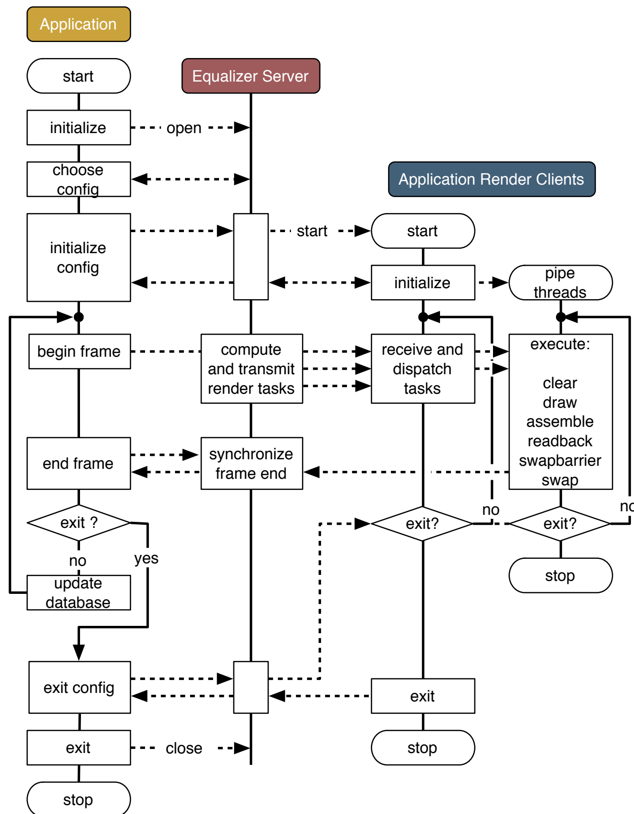



Figure 3: Simplified execution flow of an Equalizer application, omitting event handling and application-node rendering threads.

rendering. All tasks have default implementations so that only the application specific methods have to be implemented, which includes at least the *Channel::draw* method. For example, the default callbacks for frame recomposition during scalable rendering implement tile-based assembly for sort-first and stereo decompositions, and $z$-buffer compositing for sort-last rendering of polygonal data. A detailed description of all methods can be found in [16].

Event handling is implemented by listening asynchronously for events from all windows. Events are transformed from window-system specific events into generic window events, and dispatched to the correct window. The window either processes the event locally, or converts it into a config event to be send to the application node. The application node processes the config events as part of its main rendering loop. A detailed description of event handling can be found in [15].

In addition to executing the application code in the right context, the client library implements image compression and transmission, network swap barrier support and distributed object support.

## 4.4 Equalizer Server

The Equalizer server receives requests from all applications on the visualization system. It serves these requests using the application's specific configuration, launching rendering clients on the nodes, determining the rendering tasks for a frame, and synchronizing the completion of frames.

The server maintains the configurations for different applications. Maintaining the configuration on the server facilitates an extension to cross-application load balancing, resource reservation and further system-wide resource management. Each configuration, similarly to a MPK configuration, consists of two parts. The first part is a hierarchical resource description derived from the physical and logical environment of the application. The second part consists of the compound tree, which declares how the resources are used for rendering. The compounds are the heart of the scalable rendering engine, and are described in detail below.

At the top of the hierarchy are *nodes*, which represent a process, typically on a single computer of a cluster. A node has *pipes*, which are a representation of the graphic cards in a machine. All tasks for a pipe and its children are executed in a separate thread. A pipe has *windows*, which represent an OpenGL on-screen or off-screen drawable. By default, all windows of a pipe share display lists and other OpenGL objects. A window has *channels*, which embody an OpenGL viewport in a window.

Figure 4 shows a two-node, three-pipe, three-window, four-channel configuration driving a four-sided CAVE$^{\text{TM}}$. The channels declared in the resource section are used by the compounds for rendering. The leaf compounds, which execute the rendering, use a *swap barrier* to synchronize their output. The root compound specifies that the left and right eye are used for stereo rendering.

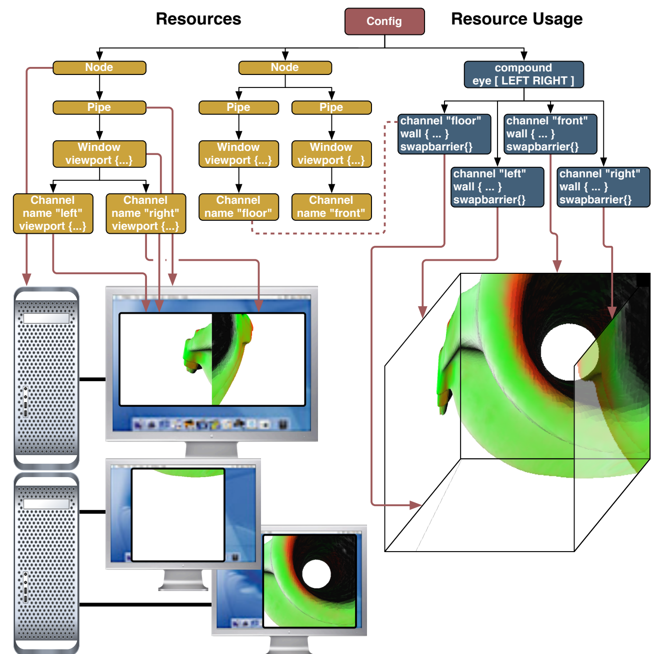



Figure 4: An example Equalizer configuration with the associated real-world counterparts.

## 4.5 Compound Trees

To configure rendering nodes and pipes for parallel rendering, Equalizer uses a *compound tree* structure similar to MPK [5]. However, the compound definition is different in a few key points to provide a more flexible and powerful configuration

First, it does not rely on a hard-coded mode which determines the task decomposition and image compositing stages. Instead, it describes the rendering and compositing tasks via the compound tree's structure.

Second, the rendering is asynchronous, and not frame-synchronized as in MPK, where all rendering threads are synchronized at the end of each frame, creating idle times for rendering threads which finish early. Equalizer removes this global synchronization point and introduces a config latency $l_{\text{config}}$, which defines how many frames the slowest rendering thread is allowed to fall behind. Hence at the end of frame $i$, the completion of frame $i - l_{\text{config}}$ will be synchronized. Note that setting $l_{\text{config}} = 0$ creates a frame-synchronicity as in MPK. Other synchronization points in Equalizer only include the completion of image transfers for compositing, and optional *swapbarriers* explicitly defined in the compound tree.

*Compounds* are a data structure for describing rendering tasks and form a tree. Each compound has *tasks* (clear, draw, assemble, readback), a *channel* which executes the tasks in the given order, and additional attributes. Note that a non-leaf compound traverses its children first before performing its own default tasks assemble and readback, while a leaf compound executes all tasks by default. A compound may provide *output frames* from the readback task to others, and request *input frames* from others for its assemble task, and output frames are linked to input frames by name. The readback or assemble tasks are only active if output or input frames have been specified, respectively. Otherwise the rendered image frame is left in-place on the current channel for further processing in a parent compound on that same channel.

All attributes and the channel are inherited from the parent compound. The *viewport*, *data range* and *eye* attributes are used to describe the decomposition of the parent's 2D viewport, database range and eye pass, respectively. *Swap barriers* can be used to synchronize the buffer swap on a group of channels, typically used for multi-screen setups such as CAVEs or display walls.

A simple sort-first compound configuration is shown in Figures 5 and 8(a). The root compound defines the viewport size of the channel and the frustum from the wall description. While the first child compound inherits the channel, the other compounds are executed on different channels. However, each defines a partial viewport, affecting its local view frustum. All leaf nodes execute the basic clear and draw tasks, and except for the first child have to readback the result into the specified output frames. The root compound executes the assemble task (sort-first tiled image compositing) once the output frames are available.

Note that the mapping to physical resources, the first part of the configuration (see also Figure 4), is omitted. Instead of using multiple pipes of an actual rendering cluster, all channels could be mapped to a single pipe, thus allowing testing complex configurations on single-pipe PC.

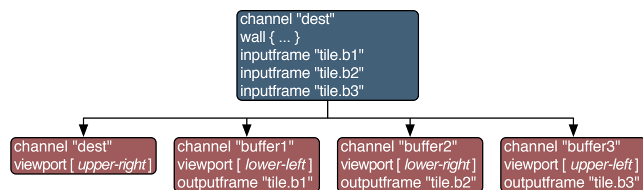



Figure 5: Compound tree for a four-to-one sort-first decomposition.

Figures 6 and 8(b) show a sort-last configuration with parallel image compositing. The leaf nodes execute the rendering, and readback of two tiles each to be $z$-composited by the other channels. The intermediate (green) compounds execute the $z$-compositing using framebuffer data from the other channels via the indicated output-input frame mapping. Once a channel has completed this assemble task (sort-last $z$-buffer image compositing) on its tile, the color framebuffer content is handed over to the root compound which puts together the tiles to form the final image. Note that a compound does not need to readback a tile which is processed in a parent node on the same channel since it is already in place (e.g. the compounds executed on the "dest" channel in Figure 6). The yellow arrows illustrate the data flow for the tile being $z$-composited by the channel named "buffer1", according to a direct-send sort-last image compositing [17].

Figures 7 and 8(c) show a mixture of decomposition algorithms in a multilevel compound tree. Stereo rendering is mixed with sort-first decomposition. The first level is a stereo decomposition for the left and right eye, which is in turn parallelized for each eye on two channels using a sort-first decomposition. The channels used for composition are also used for rendering, which again allows

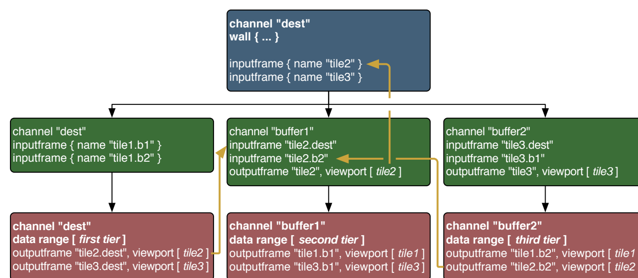



Figure 6: Compound tree for a three-to-one direct-send sort-last configuration.

some image transfer optimizations. The screenshot uses anaglyphic stereo for better readability.

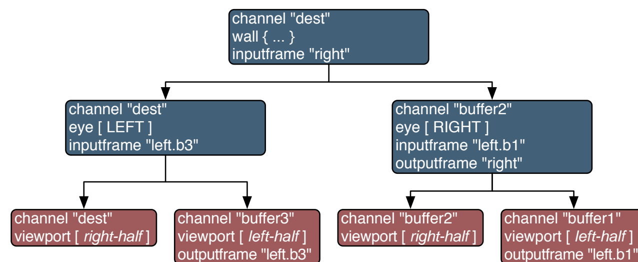



Figure 7: Compound tree for a four-to-one stereo/sort-first configuration.

Equalizer's compound description is much more flexible and powerful compared to the format used in MPK, and can be used to define parallel compositing algorithms, such as direct-send or binary-swap, as well as multilevel decompositions using different decomposition modes to balance the bottlenecks of the individual algorithms. A detailed specification can be found in [14]. Numerous example configurations are included with the Equalizer distribution.

## 5 Results

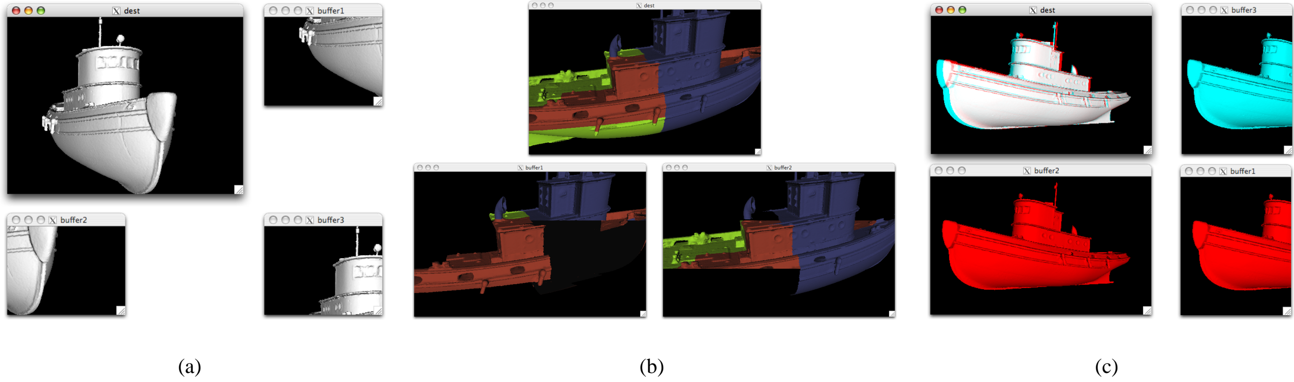
We conducted our experiments on a five node rendering cluster with the following characteristics: dual 2.2GHz AMD Opteron CPUs, 4GB of RAM, Geforce 7800 GTX PCIe graphics and a high-resolution $2560 \times 1600$ pixel LCD panel per node; 1GB network and switch. For most tests we used a destination channel with a resolution of $1280 \times 800$, since this is a more typical window size for scalable parallel rendering. Pixel read, write and network transmission performances are given in Table 1 below.

| *GL Format, Type* | *read* | *write* | *transmit* |
|---|---|---|---|
| BGRA, UNSIGNED_BYTE | 5.2$ms$ | 4.1$ms$ | 42.05$ms$ |
| DEPTH_COMPONENT, FLOAT | 5.8$ms$ | 37$ms$ | 36.41$ms$ |

Table 1: Pixel transfer timings for a full-size $1280 \times 800$ window.

Our prototype test application renders polygonal data, organized spatially in an octree for efficient view frustum culling and sort-last range selection. The data is rendered using display lists, and each vertex consist of 24 bytes (position+normal). We use a fixed camera path of 100 frames to obtain the average frames per second as the result. The model used was the Thai Statue consisting of 10M polygons from the Stanford 3D Scanning Repository.

Due to the limitations of the scope of this paper, our experimental results provide the fundamental evidence of the flexibility and scalability potential of Equalizer, but do not cover an extensive range of




(a) (b) (c)

Figure 8: (a) Sort-first tiled screen, for display walls or CAVE™setups – compound tree Figure 5. (b) Sort-last scalable rendering – compound tree Figure 6. (c) Stereo separation and sort-first decomposition – compound tree Figure 7 .

data sets, compound configurations or cluster sizes. This will have to be done in a dedicated performance study.

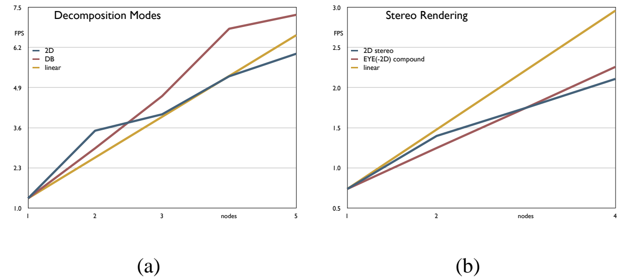
## 5.1 Decomposition Modes

The power of Equalizer lies in its flexibility to configure different scalable task decomposition and image compositing strategies efficiently using the compound tree structure. Various exemplary use cases have already been shown demonstrating the power of the compounds structure in Section 4.5, including tiled screen rendering (e.g. for display walls or CAVEs), partitioned rendering of the geometry database (mostly for scalability) or an eye-separated sort-first parallelized stereo rendering. Figure 9 demonstrates another complex setup where multiple nodes are used to drive two side-by-side projectors of a small wall display. Rendering is executed sort-last by 7 nodes while $z$-compositing is performed by 3 nodes each for the two projectors. Hence the 7 draw channels each output 6 tiles to be $z$-composited.

The quintessential benefit of Equalizer's process model and compound tree structure lies in an easy-to-configure and very scalable parallel rendering system. Therefore, we provide a brief experimental analysis of Equalizer's scalability that should demonstrate the potential of the system, while a more extensive study is beyond the scope of this paper. Equalizer sort-last image compositing scalability results can also be found in [17].

In the first benchmarks, we measured the performance of different task decomposition modes. In Figure 10(a) we use $n$-to-one sort-first and sort-last decompositions. The sort-first compounds use a trivial tile assembly on the destination channel, while the sort-last compounds use direct-send compositing. For sort-first parallel rendering, the speedup heavily depends on the decomposition of the view frustum, and hence the tiling of the window. For this study the data set is roughly placed in the middle of the screen such that a simple tiling results in a fair load distribution. The graph 2D in Figure 10(a) shows a nice close-to linear speedup for sort-first rendering, and as expected the overhead from clipped primitives is not dominating at small numbers of tiles. Equalizer also shows excellent scalability with respect to sort-last rendering, graph DB in Figure 10(a). Image compositing overhead is not manifested at this level of parallelism, partly also due to the efficient direct-send compositing algorithm (see also [17]).

The second set of benchmarks in Figure 10(b) uses different approaches to scale the performance during stereo rendering. The first graph 2D-stereo uses a sort-first decomposition where each leaf node renders two eye passes, which are assembled on the destination channel in the parent node into the correct stereo buffers. The second graph EYE-2D does first a stereo decomposition, separating into left and right eye rendering tasks, and then a sort-first decomposition into screen tiles. The graphs in Figure 10(b) show a good linear speedup, but also indicate that the more complicated stereo image assembly and compositing incurs a small overhead factor.




(a) (b)

Figure 10: (a) Sort-first and sort-last many-to-one rendering performance, and (b) different stereo rendering decompositions.

## 5.2 Latency and Viewport Size

In these benchmarks we measure the influence of the viewport size and latency on the performance. All test were conducted using a sort-last direct-send configuration with all five nodes. Figure 11(a) varies the config latency $l_{\text{config}}$ from 0 to 6. One can observe that increasing the latency from a strict frame synchronization with $l_{\text{config}} = 0$ immediately increases the performance by about 15%. This is achieved through reduced synchronization bottlenecks as rendering channels can overlap their draw tasks between frames. We also notice, as expected, that further increasing the latency does not further improve rendering performance, due to other synchronization constraints such as image transfers. We can conclude that a small latency of only one or two frames is sufficient to avoid most drawbacks of a strictly frame synchronized parallel rendering execution.

In Figure 11(a) experiments with different viewport sizes for the destination window are shown, and hence the amount of transferred and $z$-composited pixel data varies accordingly. The graph exhibits the expected asymptotic behaviour towards the constant time composition cost of direct send, as analyzed in [17], regardless of the viewport size. Since the composition cost is directly dependent on the viewport size, the performance approaches, and is limited by the constant time compositing as soon as the draw cost is reduced sufficiently by parallel load distribution. This the normal expected behavior. However, we would like to point our here that the flexible compound structure allows for complex combinations of parallel rendering and parallel compositing where the number of contributing channels can vary and thus allows for optimized resource usage.

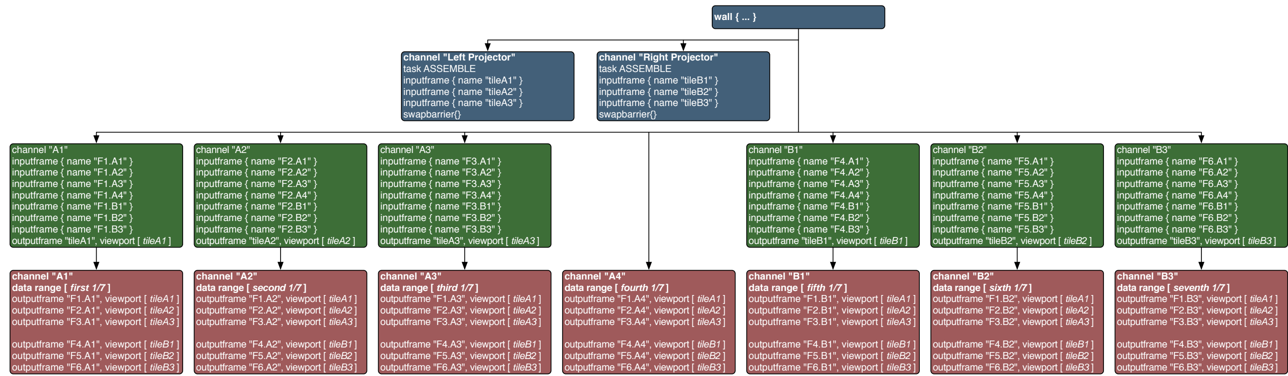
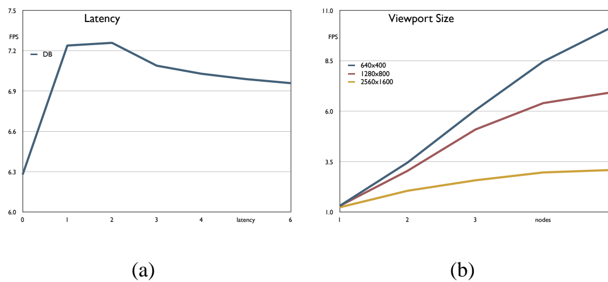



Figure 9: Two-projector wall driven by 7 sort-last rendering nodes, where $z$-compositing is done on 3 nodes for each projector.




(a) (b)

Figure 11: Influence of the (a) latency and (b) viewport size on the performance.

## 6 Discussion and Conclusion

In this paper we presented a state-of-the art parallel rendering framework, which has been written to be minimally invasive in order to facilitate the porting and development of real-world visualization applications. Equalizer has been designed to be as generic as possible to support development of parallel rendering applications for different data types.

Parallel rendering of transparent data is currently supported by sort-first configurations. For sort-last rendering of transparent surfaces or volume data, back-to-front spatial data partitioning would have to be implemented, as well as an $\alpha$-compositing compound. Scalable sort-first rendering depends on a balanced distribution of the rendering cost across the different screen tiles. To achieve this, dynamic tile decomposition must be supported as well as some basic rendering cost heuristics for effective load balancing. The above extensions pose interesting but also tractable challenges and are lined up for integration into Equalizer.

The current Equalizer implementation covers basic scalable rendering functionality. We plan to extend this functionality to include also time-multiplex (DPlex) support, sophisticated automatic loadbalancing for sort-first and sort-last task decompositions, as well as an API to compress and mask the channels' screen-frames for optimized image transport. Aside from the core parallel rendering API, we plan to improve the resource management capabilities of the server by enabling it to handle multiple applications, resource reservation and cross-application loadbalancing. Furthermore, the creation of a transparent OpenGL layer with Equalizer as the backend will allow running existing applications alongside with parallel applications. Eventually we will integrate remote visualization capabilities, for example by supporting the VNC protocol.


## Acknowledgements

We would like to thank and acknowledge the Stanford 3D Scanning Repository and Cyberware Inc. for providing the 3D geometric test data sets.


## References


[1] OpenGL Multipipe SDK.

[2] G. Agranov and C. Gotsman. Algorithms for rendering realistic terrain image sequences and their parallel implementation. *The Visual Computer*, 11(9):455–464, 1995.

[3] J. Ahrens and J. Painter. Efficient sort-last rendering using compression-based image compositing. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, 1998.

[4] J. Allard, V. Gouranton, L. Lecointre, E. Melin, and B. Raffin. Netjuggler: Running VR Juggler with multiple displays on a commodity component cluster. In *Proceeding IEEE Virtual Reality*, pages 275–276, 2002.

[5] P. Bhaniramka, P. C. D. Robert, and S. Eilemann. OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization*, pages 119–126, 2005.

[6] A. Bierbaum and C. Cruz-Neira. ClusterJuggler: A modular architecture for immersive clustering. In *Proceedings Workshop on Commodity Clusters for Virtual Reality, IEEE Virtual Reality Conference*, 2003.

[7] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A virtual platform for virtual reality application development. In *Proceedings of IEEE Virtual Reality*, pages 89–96, 2001.

[8] W. Blanke, C. Bajaj, D.Fussel, and X. Zhang. The metabuffer: A scalable multi-resolution 3-d graphics system using commodity rendering engines. Technical Report TR2000-16, University of Texas at Austin, 2000.

[9] X. Cavin and C. Mion. Pipelined sort-last rendering: Scalability, performance and beyond. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, 2006.

[10] X. Cavin, C. Mion, and A. Filbois. COTS cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In *Proceedings IEEE Visualization*, pages 111–118. Computer Society Press, 2005.

[11] W. T. Correa, J. T. Klosowski, and C. T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96, 2002.

[12] T. W. Crockett. An introduction to parallel rendering. *Parallel Computing*, 23:819–843, 1997.

[13] S. Eilemann. Equalizer. http://www.equalizergraphics.com/, 2006.

[14] S. Eilemann. Equalizer compound specification. http://www.equalizergraphics.com/documents/design/compounds.html, 2006.

[15] S. Eilemann. Equalizer event handling specification. http://www.equalizergraphics.com/documents/design/eventHandling.html, 2006.

[16] S. Eilemann. Equalizer task methods specification. http://www.equalizergraphics.com/documents/design/taskMethods.html, 2006.

[17] S. Eilemann and R. Pajarola. Direct send compositing for parallel sort-last rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, 2007.

[18] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. PixelFlow: The realization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware*, pages 57–68, 1997.

[19] A. Garcia and H.-W. Shen. An interleaved parallel volume renderer with PC-clusters. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 51–60, 2002.

[20] E. Gobbetti and F. Marton. Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(1):815–826, February 2004.

[21] M. Guthe, P. Borodin, Ä. Balazs, and R. Klein. Real-time appearance preserving out-of-core rendering with shadows. In *Proceedings Eurographics Workshop on Rendering Techniques*, pages 69–80, 2004.

[22] S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive rendering of large volume data sets. In *Proceedings IEEE Visualization*, pages 53–60. Computer Society Press, 2002.

[23] J. Huang, N. Shareef, R. Crawfis, P. Sadayappan, and K. Mueller. A parallel splatting algorithm with occlusion culling. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, 2000.

[24] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. Distributed rendering for scalable displays. *IEEE Supercomputing*, October 2000.

[25] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A scalable graphics system for clusters. In *Proceedings ACM SIGGRAPH*, pages 129–140. ACM Press, 2001.

[26] G. Humphreys and P. Hanrahan. A distributed graphics system for large tiled displays. *IEEE Visualization 1999*, pages 215–224, October 1999.

[27] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

[28] H. Igehy, G. Stoll, and P. Hanrahan. The design of a parallel graphics interface. *Proceedings of SIGGRAPH 98*, pages 141–150, July 1998.

[29] A. Johnson, J. Leigh, P. Morin, and P. Van Keken. GeoWall: Stereoscopic visualization for geoscience research and education. *IEEE Computer Graphics and Applications*, 26(6):10–14, November-December 2006.

[30] K. Jones, C. Danzer, J. Byrnes, K. Jacobson, P. Bouchaud, D. Courvoisier, S. Eilemann, and P. Robert. SGI®OpenGL Multipipe™SDK User's Guide. Technical Report 007-4239-004, Silicon Graphics, 2004.

[31] C. Just, A. Bierbaum, A. Baker, and C. Cruz-Neira. VR Juggler: A framework for virtual reality development. In *Proceedings Immersive Projection Technology Workshop*, 1998.

[32] P. G. Lever. SEPIA – applicability to MVC. White paper Manchester Visualization Centre (MVC), University of Manchester, 2004.

[33] P. P. Li, W. H. Duquette, and D. W. Curkendall. RIVA: A versatile parallel rendering system for interactive scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):186–201, 1996.

[34] P. P. Li, S. Whitman, R. Mendoza, and J. Tsiao. ParVox: A parallel splatting volume rendering system for distributed visualization. In *Proceedings IEEE Parallel Rendering Symposium*, pages 7–14, 1997.

[35] S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heirich. Scalable interactive volume rendering using off-the-shelf components. Technical Report CACR-2001-189, California Institute of Technology, 2001.

[36] S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heirich. Scalable interactive volume rendering using off-the-shelf components. In *Proceedings IEEE Symposium on Parallel and Large Data Visualization and Graphics*, pages 115–121, 2001.

[37] L. Moll, A. Heirich, and M. Shand. Sepia: scalable 3D compositing using PCI pamette. In *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 146–155, 1999.

[38] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[39] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. In *Proceedings ACM SIGGRAPH*, pages 231–240, 1992.

[40] C. Mueller. The sort-frst rendering architecture for high-performance graphics. In *Proceedings Symposium on Interactive 3D Graphics*, pages 75–84. ACM SIGGRAPH, 1995.

[41] C. Mueller. Hierarchical graphics databases in sort-first. In *Proceedings IEEE Symposium on Parallel Rendering*, pages 49–. Computer Society Press, 1997.

[42] S. Muraki, M. Ogata, K.-L. Ma, K. Koshizuka, K. Kajihara, X. Liu, Y. Nagano, and K. Shimokawa. Next-generation visual supercomputing using PC clusters with volume graphics hardware devices. In *Proceedings ACM/IEEE Conference on Supercomputing*, pages 51–51, 2001.

[43] W. Nie, J. Sun, J. Jin, X. Li, J. Yang, and J. Zhang. A dynamic parallel volume rendering computation mode based on cluster. In *Proceedings Computational Science and its Applications*, volume 3482 of *Lecture Notes in Computer Science*, pages 416–425, 2005.

[44] J. Rohlf and J. Helman. IRIS Performer: A high performance multiprocessing toolkit for real-time 3D graphics. In *Proceedings ACM SIGGRAPH*, pages 381–394. ACM Press, 1994.

[45] R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with K-way replication. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. Computer Society Press, 2001.

[46] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proceedings Eurographics Workshop on Graphics Hardware*, pages 97–108, 2000.

[47] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load balancing for multi-projector rendering systems. In *Proceedings Eurographics Workshop on Graphics Hardware*, pages 107–116, 1999.

[48] J. P. Schulze and U. Lang. The parallelization of the perspective shear-warp volume rendering algorithm. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 61–70, 2002.

[49] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A high-performance display subsystem for PC clusters. In *Proceedings ACM SIGGRAPH*, pages 141–148, 2001.

[50] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett. SLIC: Scheduled linear image compositing for parallel volume rendering. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 33–40, 2003.

[51] X. Tong, W. Wang, W. Tsang, and Z. Tang. Efficiently rendering large volume data using texture mapping hardware. In *EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, 1999.

[52] G. Vezina and P. K. Robertson. Terrain perspectives on a massively parallel SIMD computer. In *Proceedings Computer Graphics International (CGI)*, pages 163–188, 1991.

[53] C. M. Wittenbrink. Survey of parallel volume rendering algorithms. In *Proceedings Parallel and Distributed Processing Techniques and Applications*, pages 1329–1336, 1998.

[54] D.-L. Yang, J.-C. Yu, and Y.-C. Chung. Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *Journal of Supercomputing*, 18(2):201–22–, February 2001.

[55] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-VDR: Out-of-core view-dependent rendering of gigantic models. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):369–382, July-August 2005.

[56] X. Zhang, C. Bajaj, and W. Blanke. Scalable isosurface visualization of massive datasets on COTS clusters. In *Proceedings IEEE Symposium on Parallel and Large Data Visualization and Graphics*, pages 51–58, 2001.